O'REILLY®

Czysty kod Receptury

Przepisy na poprawienie struktury i jakości Twojego kodu

Helion

Maximiliano Contieri

Tytuł oryginału: Clean Code Cookbook: Recipes to Improve the Design and Quality of Your Code

Tłumaczenie: Grzegorz Werner

ISBN: 978-83-289-1421-6

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/czykor
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

ROZDZIAŁ 6.

Kod deklaratywny

Zachowanie to najważniejszy aspekt oprogramowania. Właśnie na nim polegają użytkownicy. Użytkownicy lubią, gdy dodajemy zachowania (jeśli naprawdę tego chcieli), ale kiedy zmieniamy lub usuwamy zachowania, od których są zależni (wprowadzamy usterki), przestają nam ufać.

— Michael Feathers, *Praca z zastanym kodem*

6.0. Wprowadzenie

Kod deklaratywny to kod, który opisuje, *co* program powinien zrobić, zamiast określać kroki, które program powinien podjąć w celu wykonania zadania. Kod skupia się na pożądanym wyniku (co), a nie procesie osiągania tego wyniku (jak). Kod deklaratywny jest czytelniejszy i bardziej zrozumiały niż kod imperatywny, który określa kroki podejmowane w celu wykonania zadania. Kod jest również bardziej zwięzły i skupiony na ostatecznym wyniku, a nie na konkretnych szczegółach osiągania tego wyniku.

Kod deklaratywny jest często używany w językach programowania, które obsługują programowanie funkcjonalne, czyli paradygmat programowania, który kładzie nacisk na używanie funkcji do opisania obliczeń wykonywanych przez program. Przykładem deklaratywnego języka programowania może być SQL, używany do zarządzania bazami danych, albo HTML, używany do strukturyzowania i formatowania stron internetowych.

W tworzeniu oprogramowania panuje inercja sięgająca czasów, w których ze względu na ograniczenia czasu i miejsca trzeba było pisać kod w językach niskopoziomowych. Dziś jest inaczej, ponieważ nowoczesne kompilatory i maszyny wirtualne są inteligentniejsze niż kiedykolwiek przedtem i Tobie pozostawiają ważne zadanie pisania wysokopoziomowego, deklaratywnego i czystego kodu.

6.1. Zawężanie wielokrotnie wykorzystywanych zmiennych

Problem

Używasz tej samej zmiennej w różnych zasięgach.

Rozwiązanie

Nie powinieneś zapisywać i odczytywać tej samej zmiennej w różnych celach. Powinieneś próbować określać minimalny zasięg (czas życia) wszystkich zmiennych lokalnych.

Omówienie

Wielokrotne wykorzystywanie zmiennych utrudnia śledzenie zasięgów i granic, a zarazem uniemożliwia narzędziom do refaktoryzacji wyodrębnienie niezależnych bloków kodu. Kiedy programujesz skrypt, często wielokrotnie wykorzystujesz te same zmienne. Po kilku operacjach wycięcia i wklejenia możesz otrzymać ciągłe bloki kodu. Podstawową przyczyną tego problemu jest kopiowanie kodu. Zamiast tego powinieneś zastosować recepturę 10.1, „Usuwanie powtarzalnego kodu". Praktyczna reguła jest taka, że powinieneś jak najbardziej zawężać zasięgi, ponieważ rozszerzony zasięg prowadzi do nieporozumień i utrudnia debugowanie.

W poniższej próbce kodu wielokrotnie wykorzystano zmienną `total`:

```
// Wypisujemy sumę wiersza
double total = item.getPrice() * item.getQuantity();
System.out.println("Razem: " + total);

// Wypisujemy sumę kwot
total = order.getTotal() - order.getDiscount();
System.out.println("Do zapłaty: " + total );

// Zmienna 'total' jest używana wielokrotnie
```

Powinieneś zawęzić zasięg zmiennej i podzielić kod na dwa różne bloki. Możesz to osiągnąć, jeśli wyodrębnisz je zgodnie z recepturą 10.7, „Wyodrębnianie metody do obiektu":

```
function printLineTotal() {
  double lineTotal = item.getPrice() * item.getQuantity();
  System.out.println("Razem: " + lineTotal);
}

function printAmountTotal() {
  double amountTotal = order.getTotal() - order.getDiscount();
  System.out.println("Do zapłaty: " + amountTotal);
}
```

Ogólna zasada jest taka, że powinieneś unikać wielokrotnego wykorzystywania nazw zmiennych. Używaj bardziej lokalnych, specyficznych nazw, które ujawniają Twoje intencje.

Powiązane receptury

Receptura 10.1, „Usuwanie powtarzalnego kodu”

Receptura 10.7, „Wyodrębnianie metody do obiektu”

Receptura 11.1, „Dzielenie zbyt długich metod”

Receptura 11.3, „Ograniczanie nadmiaru zmiennych”

Ujawnianie intencji

Kod **ujawniający intencje** jasno komunikuje swoje przeznaczenie innym deweloperom, którzy mogą w przyszłości go czytać lub z nim pracować. Kod ujawniający intencje jest bardziej behawioralny, deklaratywny, czytelny, zrozumiały i łatwiejszy w konserwacji.

6.2. Usuwanie pustych wierszy

Problem

Masz duże fragmenty kodu rozdzielone pustymi wierszami.

Rozwiązanie

Użyj receptury 10.7, „Wyodrębnianie metody do obiektu”, aby wydzielić bloki działań identyfikowane przez puste wiersze.

Omówienie

Krótsze funkcje zwiększają czytelność, ułatwiają ponowne wykorzystanie kodu i są zgodne z zasadą KISS. Oto przykład, w którym możesz zgrupować ciągłe fragmenty kodu oddzielone pustymi wierszami:

```
function translateFile() {
    $this->buildFilename();
    $this->readFile();
    $this->assertFileContentsOk(); // znacznie więcej wierszy

    // puste miejsce jako pauza w definicji
    $this->translateHyperlinks();
    $this->translateMetadata();
    $this->translatePlainText();

    // Kolejne puste miejsce
    $this->generateStats();
    $this->saveFileContents(); // Znacznie więcej wierszy
}
```

Możesz zmienić ten kod zgodnie z recepturą 10.7, „Wyodrębnianie metody do obiektu”, aby uzyskać krótszą wersję przez zgrupowanie fragmentów:

```
function translateFile() {
    $this->readFileToMemory();
    $this->translateContents();
    $this->generateStatsAndSaveFileContents();
}
```

Jeśli używasz lintera, możesz skonfigurować go tak, aby ostrzegał Cię o pustych wierszach i zbyt długich metodach. Puste wiersze są nieszkodliwe, ale dają okazję do podziału kodu na mniejsze porcje. Jeśli zamiast (albo oprócz) tego dzielisz kod komentarzami, jest to brzydki zapach kodu, który prosi się o refaktoryzację (patrz receptura 8.6, „Usuwanie komentarzy wewnątrz metod").

Zasada KISS

KISS to skrót od ang. „Keep It Simple, Stupid" (nie komplikuj, głupcze!). Mówi ona, że systemy działają najlepiej, kiedy dba się o ich prostotę, zamiast je niepotrzebnie komplikować. Prostsze systemy są bardziej zrozumiałe oraz łatwiejsze w użyciu i konwersacji niż systemy skomplikowane, a przez to rzadziej zawodzą lub generują nieoczekiwane wyniki.

Powiązane receptury

Receptura 8.6, „Usuwanie komentarzy wewnątrz metod"

Receptura 10.7, „Wyodrębnianie metody do obiektu"

Receptura 11.1, „Dzielenie zbyt długich metod"

Zobacz również

Receptura ta jest szczegółowo wyjaśniona w książce *Czysty kod* Roberta Martina.

6.3. Usuwanie wersjonowanych metod

Problem

Masz metody oznaczone informacjami o wersji, takie jak `sort`, `sortOld`, `sort20210117`, `sortFirst➥Version`, `workingSort` itd.

Rozwiązanie

Usuń informacje o wersji z nazw metod i zamiast tego używaj oprogramowania do kontroli wersji.

Omówienie

Wersjonowane funkcje zmniejszają czytelność kodu i utrudniają jego konserwację. Powinieneś mieć tylko jedną roboczą wersję swojego artefaktu (klasy, metody, atrybutu), a kwestie chronologii pozostawić w gestii systemu kontroli wersji. Jeśli masz kod, który używa wersjonowanych metod, takich jak poniższe:

```
findMatch()
findMatch_new()
findMatch_newer()
findMatch_newest()
findMatch_version2()
findMatch_old()
findMatch_working()
findMatch_for_real()
findMatch_20200229()
findMatch_thisoneisnewer()
findMatch_themostnewestone()
findMatch_thisisit()
findMatch_thisisit_for_real()
```

…powinieneś zastąpić wszystkie wystąpienia prostszym:

```
findMatch()
```

Podobnie jak w przypadku innych wzorców, możesz opracować wewnętrzne zasady i jasno zakomunikować je innym; możesz też dodać automatyczne reguły do znajdowania wersjonowanych metod za pomocą wyrażeń regularnych. Tworząc oprogramowanie, zawsze trzeba zarządzać czasem i ewolucją kodu. Na szczęście dziś masz dojrzałe narzędzia, które rozwiązują ten problem.

System kontroli kodu źródłowego

System kontroli kodu źródłowego to narzędzie, które pozwala deweloperom śledzić zmiany dokonane w kodzie źródłowym oprogramowania. Możesz jednocześnie pracować z wieloma innymi deweloperami nad tą samą bazą kodu z możliwością cofania zmian i zarządzania różnymi wersjami kodu. Obecnie najbardziej popularnym systemem kontroli kodu źródłowego jest Git.

Powiązane receptury

Receptura 8.5, „Przekształcanie komentarzy w nazwy funkcji”

6.4. Usuwanie podwójnych zaprzeczeń

Problem

Masz metodę, której nazwa wywodzi się z negatywnego warunku, i chcesz sprawdzić, czy ten warunek nie jest spełniony.

Rozwiązanie

Zawsze nadawaj pozytywne nazwy swoim zmiennym, metodom i klasom.

Omówienie

Ta receptura ma na celu zwiększenie czytelności kodu; kiedy czytasz negatywne warunki, łatwo o nieporozumienie. Oto przykład podwójnego zaprzeczenia:

```
if (!work.isNotFinished())
```

Zmień je na wersję pozytywną:

```
if (work.isDone())
```

Możesz nakazać swojemu linterowi, aby ostrzegał Cię przed wyrażeniami takimi jak `!not` albo `!isNot`. Zaufaj swoim testom pokrycia i korzystaj z bezpiecznych zmian nazw oraz innych refaktoryzacji.

Powiązane receptury

Receptura 10.4, „Usuwanie z kodu pomysłowości”

Receptura 14.3, „Reifikacja zmiennych logicznych”

Receptura 14.11, „Zapobieganie zwracaniu wartości logicznych podczas sprawdzania warunków”

Receptura 24.2, „Radzenie sobie z wartościami pseudoprawdziwymi”

Zobacz również

Artykuł *Remove Double Negative* w witrynie Refactoring.com (*https://oreil.ly/bR1Sf*)

6.5. Zmienianie źle przypisanych obowiązków

Problem

Masz metody w niewłaściwych obiektach.

Rozwiązanie

Utwórz lub przeciąż właściwe obiekty, aby znaleźć odpowiednie miejsce dla metod zgodnie z modelem MAPPER.

Omówienie

Znajdowanie odpowiedzialnych obiektów to trudne zadanie. Musisz odpowiedzieć na pytanie: „Czyim obowiązkiem jest…?”. Jeśli porozmawiasz z kimś spoza świata oprogramowania, prawdopodobnie będzie mógł Ci udzielić wskazówek, gdzie powinieneś umieścić poszczególne obowiązki. Natomiast inżynierowie oprogramowania mają tendencję do umieszczania obowiązków w dziwnych miejscach… takich jak metody pomocnicze!

Oto przykład z odpowiedzialnością za dodawanie:

```
Number>>#add: a to: b
  ^ a + b
// Jest to naturalne w wielu językach programowania, ale nienaturalne w prawdziwym życiu
```

Oto inne podejście:

```
Number>>#add: adder
  ^ self + adder
```

```
// Nie skompiluje się to w niektórych językach programowania,
// ponieważ zwykle zabraniają one zmiany niektórych zachowań klas podstawowych.
// Jest to jednak właściwe miejsce dla obowiązku "dodawania"
```

Jest kilka języków, w których możesz dodawać obowiązki do typów podstawowych, a jeśli dodasz obowiązki do właściwego obiektu, z pewnością znajdziesz je w tym samym miejscu. Oto inny przykład, z definicją stałej `PI`:

```
class GraphicEditor {
  constructor() {
    this.PI = 3.14;
    // Nie powinieneś definiować stałej w tym miejscu
  }

  pi() {
    return this.PI;
    // Nie jest to obowiązek tego obiektu
  }

  drawCircle(radius) {
    console.log("Rysuję okrąg o promieniu ${radius} " +
    "i obwodzie " + (2 * this.pi()) * radius");
  }
}
```

Jeśli przeniesiesz ten obowiązek do obiektu `RealConstants`, unikniesz powtarzalnego kodu:

```
class GraphicEditor {
  drawCircle(radius) {
    console.log("Rysuję okrąg o promieniu " + radius +
      " i obwodzie " + (2 * RealConstants.pi() * radius));
  }
}
// Definiowanie PI jest obowiązkiem obiektu RealConstants (albo Number lub podobnego)

class RealConstants {
  pi() {
    return 3.14;
  }
}
```

Powiązane receptury

Receptura 7.2, „Zmienianie nazw oraz dzielenie klas pomocniczych i narzędziowych”

Receptura 17.8, „Zapobieganie zawiści funkcjonalnej”

6.6. Zastępowanie jawnych iteracji

Problem

Prawdopodobnie poznałeś pętle, kiedy uczyłeś się programować. Ale enumeratory i iteratory to narzędzia nowej generacji, a Ty potrzebujesz wyższego poziomu abstrakcji.

Rozwiązanie

Podczas iteracji nie używaj indeksów. Preferuj kolekcje wyższego poziomu.

Omówienie

Indeksy często naruszają enkapsulację i są mniej deklaratywne. Jeśli Twój język obsługuje takie konstrukcje, powinieneś faworyzować instrukcję `foreach()` albo iteratory wyższego rzędu; aby ukrywać szczegóły implementacyjne, możesz też używać instrukcji `yield()`, *pamięci podręcznych*, *pośredników*, *leniwego wczytywania* i wielu innych rozwiązań.

Oto przykład ze strukturalną iteracją opartą na indeksie `i`:

```
for (let i = 0; i < colors.length; i++) {
  console.log(colors[i]);
}
```

Poniższy kod jest bardziej deklaratywny i wysokopoziomowy:

```
colors.forEach((color) => {
  console.log(color);
});

// Używasz domknięć i funkcji strzałki
```

Istnieją pewne wyjątki. Jeśli dziedzina problemu wymaga bijekcji elementów (zgodnie z definicją w rozdziale 2.) na liczby naturalne, takie jak indeksy, pierwsze rozwiązanie jest zadowalające. Pamiętaj, aby zawsze znajdować analogie ze świata rzeczywistego. Ten rodzaj brzydkiego zapachu nie alarmuje wielu deweloperów, ponieważ uważają oni to za nieistotny szczegół, ale w czystym kodzie chodzi właśnie o takie deklaratywne szczegóły, które robią różnicę.

Powiązane receptury

Receptura 7.1, „Rozwijanie skrótów”

6.7. Dokumentowanie decyzji projektowych

Problem

Podjąłeś niebanalne decyzje dotyczące kodu i musisz udokumentować przyczyny.

Rozwiązanie

Używaj deklaratywnych nazw, które ujawniają intencje.

Omówienie

Musisz dbać o deklaratywność swoich decyzji projektowych lub implementacyjnych, na przykład przez wyodrębnianie decyzji i nadawanie im klarownych nazw, które ujawniają intencje. Nie powinieneś używać komentarzy do kodu, ponieważ komentarze to „martwy kod” i mogą łatwo się

zdezaktualizować, w dodatku w ogóle nie są kompilowane. Po prostu jawnie wyraź swoją decyzję albo przekształć komentarz w metodę. Czasem trafiają się arbitralne reguły, których nie da się łatwo przetestować. Jeśli na przykład nie możesz napisać testu wykrywającego ewentualny błąd, zamiast komentarza utwórz funkcję o przejrzystej i deklaratywnej nazwie, która ostrzega przed przyszłymi zmianami.

Oto przykład niejawnej decyzji projektowej:

```
// Ten proces trzeba uruchamiać z większą ilością pamięci
set_memory("512k");

run_process();
```

Poniższy kod jest jawny i klarowny, ponieważ wskazuje przyczynę zwiększenia ilości pamięci:

```
increase_memory_to_avoid_false_positives();
run_process();
```

Kod to proza. A decyzje projektowe powinny być narracją.

Powiązane receptury

Receptura 8.5, „Przekształcanie komentarzy w nazwy funkcji”

Receptura 8.6, „Usuwanie komentarzy wewnątrz metod”

6.8. Zastępowanie magicznych liczb stałymi

Problem

Masz metodę, która dokonuje obliczeń na mnóstwie liczb bez opisywania ich semantyki.

Rozwiązanie

Unikaj *magicznych, niewyjaśnionych liczb*. Nie znasz ich źródła i powinieneś bardzo uważać, kiedy je zmieniasz, żeby nie popsuć kodu.

Omówienie

Magiczne liczby są źródłem sprzężenia. Są nieczytelne i trudno się je testuje. Powinieneś nadać każdej stałej semantyczną nazwę (znaczącą i ujawniającą intencję) oraz zastąpić je parametrami, żeby móc pozorować je z zewnątrz (patrz receptura 20.4, „Zastępowanie atrap prawdziwymi obiektami”). Definicja stałej jest często innym obiektem niż użytkownik stałej, a na szczęście wiele linterów potrafi wykrywać literały liczbowe w atrybutach i metodach.

Oto dobrze znana stała:

```
function energy($mass) {
    return $mass * (299792 ** 2);
}
```

Możesz zmodyfikować ten kod w następujący sposób:

```
function energy($mass) {
    return $mass * (LIGHT_SPEED_KILOMETERS_OVER_SECONDS ** 2);
}
```

Powiązane receptury

Receptura 5.2, „Deklarowanie zmiennych jako zmiennych”

Receptura 5.6, „Zamrażanie mutowalnych stałych”

Receptura 10.4, „Usuwanie z kodu pomysłowości”

Receptura 11.4, „Usuwanie nadmiaru nawiasów”

Receptura 17.1, „Ujawnianie ukrytych założeń”

Receptura 17.3, „Rozbijanie boskich obiektów”

6.9. Oddzielanie „co” od „jak”

Problem

Masz kod, który przygląda się wewnętrznym trybom zegara, zamiast patrzeć na jego wskazówki.

Rozwiązanie

Nie majstruj przy szczegółach implementacyjnych. Bądź deklaratywny, a nie imperatywny.

Omówienie

Trzeba uważnie wybierać nazwy, aby uniknąć przypadkowego sprzężenia. Oddzielanie obowiązków bywa trudnym zadaniem w branży oprogramowania, ale oprogramowanie funkcjonalne potrafi przejść próbę czasu. Natomiast oprogramowanie implementacyjne wprowadza sprzężenia i trudniej je zmienić.

Czasem zmiany są dokumentowane z wykorzystaniem komentarzy, ale nie jest to dobre rozwiązanie, ponieważ komentarze rzadko się aktualizuje (patrz receptura 8.5, „Przekształcanie komentarzy w nazwy funkcji”). Jeśli faworyzujesz projektowanie pod kątem zmian i intencji, Twój kod przetrwa dłużej i będzie funkcjonował lepiej.

W poniższej próbce kodu przejście do następnego etapu jest sprzężone z oczekującym zadaniem we właściwości `stepWork`:

```
class Workflow {
    moveToNextTransitiogn() {
        // Sprzęgasz regułę biznesową z akcydentalną implementacją
        if (this.stepWork.hasPendingTasks()) {
            throw new Error('Warunki wstępne jeszcze nie są spełnione...');
        } else {
```

```
            this.moveToNextStep();
        }
    }
}
```

Oto lepsze rozwiązanie, wykorzystujące tę recepturę:

```
class Workflow {
    moveToNextTransition() {
        if (this.canMoveOn()) {
            this.moveToNextStep();
        } else {
            throw new Error(' Warunki wstępne jeszcze nie są spełnione...');
        }
    }

    canMoveOn() {
        // Ukrywasz akcydentalną implementację ("jak")
        // pod tym, jakie zadanie wykonuje kod ("co")
        return !this.stepWork.hasPendingTasks();
    }
}
```

Musisz wybierać dobre nazwy, a w razie potrzeby dodawać pośrednie warstwy, aby uniknąć przedwczesnej optymalizacji (patrz rozdział 16., „Przedwczesna optymalizacja"). Argument, że tracisz zasoby obliczeniowe, jest nieistotny. Poza tym każda nowoczesna maszyna wirtualna może umieścić te dodatkowe wywołania w pamięci podręcznej albo wpleść je w kod głównego programu.

Powiązane receptury

Receptura 8.5, „Przekształcanie komentarzy w nazwy funkcji"

Receptura 19.6, „Zmienianie nazw izolowanych klas"

6.10. Dokumentowanie wyrażeń regularnych

Problem

Masz magiczne wyrażenia regularne, które są niezrozumiałe.

Rozwiązanie

Podziel skomplikowane wyrażenia regularne na krótsze i bardziej deklaratywne przykłady.

Omówienie

Wyrażenia regularne utrudniają czytanie, konserwowanie i testowanie kodu; powinieneś używać ich tylko do walidacji łańcuchów. Jeśli musisz manipulować obiektami, niech nie będą to łańcuchy; utwórz małe obiekty zgodnie z recepturą 4.1, „Tworzenie małych obiektów".

Oto przykład wyrażenia regularnego, które nie jest deklaratywne:

```
val regex = Regex("A\\+(?:[0-9][- -]?){6,14}[0-9a-zA-Z]$")
```

Oto deklaratywna wersja, która jest bardziej zrozumiała i łatwiejsza do przetestowania:

```
val prefix = "\\+"
val digit = "[0-9]"
val space = "[- -]"
val phoneRegex = Regex("A$prefix(?:$digit$space?){6,14}$digit$")
```

Wyrażenia regularne to użyteczne narzędzie. Nie ma wielu zautomatyzowanych sposobów wykrywania potencjalnych nadużyć; pomocna może być lista zezwoleń. Są też doskonałym narzędziem do walidacji łańcuchów. Musisz używać ich w sposób deklaratywny i tylko do przetwarzania łańcuchów. Dobre nazwy pomagają zrozumieć znaczenie wzorców. Jeśli musisz manipulować obiektami lub hierarchiami, powinieneś robić to za pomocą obiektów, chyba że masz rozstrzygające wyniki testów, które pokazują *imponujący* wzrost wydajności.

Powiązane receptury

Receptura 4.7, „Reifikacja walidacji łańcuchów”

Receptura 10.4, „Usuwanie z kodu pomysłowości”

Receptura 16.2, „Usuwanie przedwczesnej optymalizacji”

Receptura 25.4, „Zastępowanie szkodliwych wyrażeń regularnych”

6.11. Przekształcanie warunków Yody

Problem

Testujesz oczekiwane wartości po lewej stronie wyrażenia.

Rozwiązanie

Pisz warunki ze zmienną po lewej stronie i testowaną wartością po prawej stronie.

Omówienie

Większość programistów pisze najpierw zmienną lub warunek, a potem testowaną wartość. W rzeczywistości jest to poprawna kolejność asercji. W niektórych językach preferuje się odwrotną kolejność, aby uniknąć przypadkowego przypisania, które może powodować logiczne błędy w kodzie.

Oto przykład warunku Yody:

```
if (42 == answerToLifeMeaning) {
  // Zapobiega przypadkowej literówce powodującej przypisanie,
  // ponieważ '42 = answerToLifeMeaning' to niepoprawne wyrażenie,
  // ale wyrażenie 'answerToLifeMeaning = 42' jest poprawne
}
```

Oto jak powinien wyglądać ten kod:

```
if (answerToLifeMeaning == 42) {
  // Można pomylić z answerToLifeMeaning = 42
}
```

Zawsze sprawdzaj, czy nie masz stałych wartości po lewej stronie porównania.

Powiązane receptury

Receptura 7.15, „Zmienianie nazw argumentów według roli”

6.12. Usuwanie dowcipnych metod

Problem

Masz kod, który może obrazić inne osoby.

Rozwiązanie

Nie bądź nieformalny ani obraźliwy. Bądź uprzejmy dla swojego kodu i czytelników.

Omówienie

Pisz kod w sposób profesjonalny, używając znaczących nazw. Twój zawód ma kreatywną stronę. Czasem może Ci się nudzić i postanowisz sobie pożartować ze szkodą dla czytelności kodu i Twojej reputacji. Oto przykład nieprofesjonalnego kodu:

```
function erradicateAndMurderAllCustomers();
// Nieprofesjonalna i obraźliwa
```

Oto bardziej profesjonalna wersja metody:

```
function deleteAllCustomers();
// Bardziej deklaratywna i profesjonalna
```

Możesz sporządzić listę zabronionych i wulgarnych słów i wyszukiwać je automatycznie albo podczas recenzowania kodu. Konwencje nazewnicze powinny mieć charakter ogólny i nie powinny obejmować żargonu kulturowego. Kod produkcyjny powinieneś pisać w sposób, który gwarantuje, że przyszli deweloperzy (a nawet przyszły Ty) łatwo go zrozumieją.

Powiązane receptury

Receptura 7.7, „Zmienianie abstrakcyjnych nazw”

6.13. Unikanie piekła wywołań zwrotnych

Problem

Masz asynchroniczny kod używający wywołań zwrotnych, który jest nadmiernie zagnieżdżony, nieczytelny i trudny w konserwacji.

Rozwiązanie

Nie używaj wywołań zwrotnych. Napisz sekwencję.

Omówienie

Z piekłem wywołań zwrotnych masz do czynienia wtedy, gdy Twój kod ma wiele zagnieżdżonych wywołań zwrotnych, przez co struktura kodu jest skomplikowana i nieczytelna. Często zdarza się to w JavaScripcie, kiedy używa się programowania asynchronicznego i przekazuje wywoływane zwrotnie funkcje jako argumenty innych funkcji. Głębokie zagnieżdżenie prowadzi do kodu, który określa się mianem Piramidy Zagłady.

Kiedy wywołujesz wewnętrzną funkcję, może ona zwracać funkcję przyjmującą wywołanie zwrotne, co prowadzi do łańcucha zagnieżdżonych wywołań zwrotnych, które trudno prześledzić i zrozumieć.

Oto krótki przykład piekła wywołań zwrotnych:

```
asyncFunc1(function (error, result1) {
  if (error) {
      console.log(error);
  } else {
      asyncFunc2(function (error, result2) {
        if (error) {
            console.log(error);
        } else {
        asyncFunc3(function (error, result3) {
           if (error) {
             console.log(error);
           } else {
           // Ciąg dalszy zagnieżdżonych wywołań zwrotnych...
          }
        });
      }
    });
  }
});
```

Możesz zapisać to w ten sposób:

```
function asyncFunc1() {
  return new Promise((resolve, reject) => {
      // operacja asynchroniczna
      // ...

      // w przypadku sukcesu
      resolve(result1);
```

```
        // w przypadku błędu
        reject(error);
    });
}

function asyncFunc2() {
    return new Promise((resolve, reject) => {
        // operacja asynchroniczna
        // ...

        // w przypadku sukcesu
        resolve(result2);

        // w przypadku błędu
        reject(error);
    });
}

async function performAsyncOperations() {
    try {
        const result1 = await asyncFunc1();
        const result2 = await asyncFunc2();
        const result3 = await asyncFunc3();

      // dalsze operacje
    } catch (error) {
        console.log(error);
    }
}

performAsyncOperations();
```

Możesz rozwiązać ten problem z wykorzystaniem obietnic oraz modelu async/await, dzięki czemu kod będzie bardziej czytelny i łatwiejszy w debugowaniu.

Powiązane receptury

Receptura 10.4, „Usuwanie z kodu pomysłowości”

Receptura 14.10, „Poprawianie zagnieżdżonego kodu strzałkowego”

6.14. Generowanie dobrych komunikatów o błędach

Problem

Musisz tworzyć dobre opisy błędów, zarówno na użytek deweloperów, którzy korzystają z Twojego kodu (i swój własny), jak i dla użytkowników końcowych.

Rozwiązanie

Używaj znaczących opisów i sugeruj opcje rozwiązania problemu. Okazanie użytkownikom takiej życzliwości będzie bardzo mile widziane.

Omówienie

Programiści rzadko są ekspertami od „wrażeń użytkownika” (UX). Pomimo to powinieneś używać deklaratywnych komunikatów o błędach z myślą o użytkownikach końcowych i pokazywać te komunikaty z jasnymi opcjami wyjścia. Przestrzegaj w odniesieniu do swoich użytkowników zasady minimalnego zaskoczenia (patrz receptura 5.6, „Zamrażanie mutowalnych stałych”).

Oto zły opis błędu:

```
alert("Anulować spotkanie?", "Tak", "Nie");

// Brak konsekwencji i akcji
// Opcje nie są jasne
```

Możesz zmienić to w bardziej deklaratywny komunikat o błędzie:

```
alert("Anulować spotkanie? \n" +
      "Utracisz całą historię zmian",
      "Anuluj spotkanie",
      "Kontynuuj edytowanie");

// Konsekwencje są klarowne
// Opcje mają kontekst
```

Nie maskuj błędów z wykorzystaniem poprawnych wartości dziedzinowych i jasno odróżniaj zero od błędu. Przyjrzyj się poniższemu kodowi, który ukrywa błąd sieciowy i niepoprawnie pokazuje zerowe saldo, sprawiając, że użytkownik końcowy wpada w panikę:

```
def get_balance(address):
    url = "https://blockchain.info/q/addressbalance/" + address
    response = requests.get(url)
    if response.status_code == 200:
        return response.text
    else:
        return 0
```

Ta wersja jest klarowniejsza i bardziej jawna:

```
def get_balance(address):
    url = "https://blockchain.info/q/addressbalance/" + address
    response = requests.get(url)
    if response.status_code == 200:
        return response.text
    else:
        raise BlockchainNotReachableError("Błąd komunikacji z blockchainem")
```

Deklaratywne opisy wyjątków

Opisy wyjątków powinny wspominać o regule biznesowej, a nie o błędzie. Dobry opis: „Liczba powinna wynosić od 1 do 100”. Zły opis: „Liczba poza granicami”. Jakie są granice?

Podczas recenzowania kodu musisz czytać wszystkie komunikaty o wyjątkach, a kiedy zgłaszasz wyjątki albo pokazujesz komunikaty — myśleć o użytkownikach końcowych.

Powiązane receptury

Receptura 15.1, „Tworzenie obiektów null”

Receptura 17.13, „Usuwanie kodu biznesowego z interfejsu użytkownika”

Receptura 22.3, „Poprawianie wyjątków pod kątem oczekiwanych przypadków”

Receptura 22.5, „Zastępowanie kodów zwrotnych wyjątkami”

6.15. Unikanie magicznych poprawek

Problem

Masz zdania, które w części języków są poprawne i magiczne, ale musisz pisać bardziej jawny kod i kierować się zasadą szybkiego reagowania na błędy.

Rozwiązanie

Usuń ze swojego kodu magiczne poprawki.

Omówienie

Niektóre języki zamiatają problemy pod dywan i dokonują magicznych poprawek i niejasnych rzutowań, czym naruszają zasadę szybkiego reagowania na błędy. Powinieneś zadbać o jawność i usunąć wszelkie niejednoznaczności. Powinieneś zmienić magiczne zdania, takie jak pokazano poniżej:

```
new Date(31, 02, 2020);

1 + 'Hello';

!3;

// W większości języków jest to poprawne
```

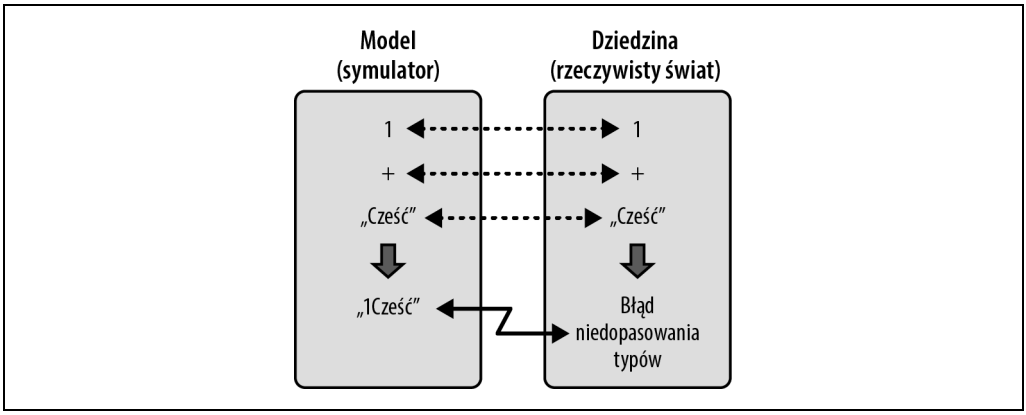
Oto jawne rozwiązanie:

```
new Date(31, 02, 2020);
// Zgłoś wyjątek

1 + 'Hello';
// niedopasowanie typów

!3;
// Negacja to operacja logiczna
```

Na rysunku 6.1 pokazano nieoczekiwany wynik dodawania liczby do łańcucha, co nie jest poprawną operacją w rzeczywistym świecie i powinno spowodować wyjątek.



Rysunek 6.1. Wykonanie metody „+" daje inne wyniki w modelu i w rzeczywistym świecie

Źródłem wielu z tych problemów są same języki. Powinieneś dbać o deklaratywność i jawność; nie nadużywaj kcydentalnych rozwiązań, zwłaszcza jeśli wydają się magiczne (w odróżnieniu od racjonalnych). Wielu programistów lubuje się w wykorzystywaniu osobliwości języków; piszą niepotrzebnie skomplikowany kod, który próbuje być pomysłowy, zamiast czystego kodu.

Powiązane receptury

Receptura 10.4, „Usuwanie z kodu pomysłowości"

Receptura 24.2, „Radzenie sobie z wartościami pseudoprawdziwymi"

Oto książka wsparta głęboką teorią i wieloma praktycznymi przykładami pisania czystego kodu!

Daniel Moka, inżynier oprogramowania, Moka IT

Funkcjonalność oprogramowania nieustannie się poszerza, a nowoczesny kod jest coraz częściej generowany przez narzędzia oparte na sztucznej inteligencji. W tych warunkach czystość kodu staje się niezwykle ważna. Niezależnie od tego, czy pracujesz na oprogramowaniu zastrzeżonym, czy otwartym, czysty kod jest najlepszym sposobem na to, aby zachować świeżość projektów i ich gotowość do rozwoju.

Receptury zawarte w tym zbiorze pozwalają nie tylko zrozumieć koncepcję czystego kodu, ale również poznać zasady identyfikacji kodu wymagającego poprawy i oceny wpływu refaktoryzacji na kod produkcyjny. Poza recepturami opisano tu potrzebne narzędzia i przedstawiono wskazówki dotyczące technik zmieniania kodu – wraz z licznymi przykładami. Omówiono ponadto różne praktyki, heurystyki i reguły projektowania oprogramowania w sposób zapewniający jego niezawodność, łatwość testowania, bezpieczeństwo i skalowalność. Docenisz, że przykłady kodu zostały przedstawione w kilku nowoczesnych językach programowania. Dzięki temu receptury zawarte w tym przewodniku są przydatne niezależnie od używanego języka.

W tej książce doskonale uchwycono głęboką wiedzę o tworzeniu oprogramowania!

Alex Bunardzic, deweloper i instruktor

W książce:

- znaczenie czystego kodu i identyfikacja możliwości jego poprawy
- techniki refaktoryzacji
- zestaw przykładów w kilku współczesnych językach programowania
- brzydkie zapachy kodu, ich konsekwencje i potencjalne rozwiązania
- techniki pisania prostego, czytelnego kodu

Maximiliano Contieri jest programistą i wykładowcą na Uniwersytecie w Buenos Aires. Pisze na popularnych platformach blogowych o czystym kodzie, refaktoryzacji i brzydkich zapachach kodu. Jest zwolennikiem stosowania fundamentalnych zasad programowania do konstruowania eleganckich, skalowalnych i solidnych rozwiązań.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1421-6
9 788328 914216
Cena: 89,00 zł